

# Practical SIMD acceleration with Boost.SIMD



Pierre Estérie  
Mathias Gaunard  
Joël Falcou

Jean-Thierry Lapresté

LRI - Université Paris Sud 11

Meeting Cpp 2012

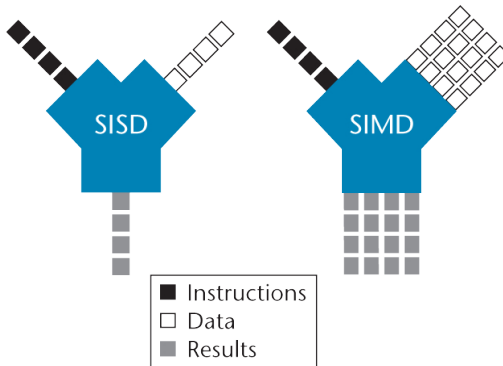
# Introduction

---

## Snapshot of the Parallel tools world

- Today's architectures around us:
  - Multicore architectures
  - Distributed architectures
  - Many Core architectures
  
- Ok but, your single core is calling you and reminds you:
  - I have a **SIMD instructions set!**

# What's SIMD?



## Principles

- Single Instruction, Multiple Data
- Operations applied on  $N \times T$  elements within a single register
- Up to  $N$  times faster than regular ALU/FPU

# Why is SIMD abstraction needed?

---

## x86 family

- MMX 64-bit float, double
- SSE 128-bit float
- SSE2 128-bit int8, int16, int32, int64, double
- SSE3
- SSSE3
- SSE4a (AMD only)
- SSE4.1
- SSE4.2
- AVX 256-bit float, double
- FMA4 (AMD only)
- XOP (AMD only)
- FMA3

## PowerPC family

- AltiVec 128-bit int8, int16, int32, int64, float
- Cell SPU 128-bit int8, int16, int32, int64, float, double

## ARM family

- VFP 64-bit float, double
- NEON 64-bit and 128-bit double, float, int8, int16, int32, int64

# Why is SIMD abstraction needed?

---

## x86 family

- New microarchitecture **Intel Haswell with AVX2**
  - 256 bits SIMD Unit
  - Customized integer intrinsics
  - scatter/gather
- New microarchitecture **Intel MIC**
  - 512 bits SIMD Unit

## PowerPC family

- AltiVec VMX128 in the **XBOX 360**
- **Power7** AltiVec with int64 and double

## ARM family

- **NEON2** NEON + double

# Why not let the compiler do it?

---

## Compilers are only so smart

- Automatic vectorization can only happen if:
  - Memory is well agenced
  - Code is inherently vectorizable
- Compiled functions are not vectorized
- Compilers don't always have enough static information to know what they can vectorize

## Conclusion

- Explicit SIMD parallelism is still the good way to get your code vectorized
- High maintainance cost for each new extension without performance loss

# What are we proposing?

---

## High Level Abstraction

- Embedded Domain Specific Language (EDSL)
  - SIMD register abstraction as data pack
  - Expression level gives wide optimizations opportunities

## STL integration

- Writing generic C++ code
- Reuse Container - Iterator - Algorithm abstraction
- Blend with modern C++ Concepts (Range...)

## How can we introduced SIMD here?

- Keep a STL based interface.
- Write SIMD code like scalar code.

# Talk Layout

---

Introduction

Interface

boost.simd and the Standard Library

Results

Conclusion



# Writing it by hand

---

Doing  $a * b + c$  with vectors of 32-bit integers : SSE4.1

```
__m128i a, b, c, result;  
result = _mm_mullo_epi32(a, _mm_add_epi32(b, c));
```

Doing  $a * b + c$  with vectors of 32-bit integers : Altivec

```
__vector int a, b, c, result;  
result = vec_cts(vec_madd( vec_ctf(a,0)  
                           , vec_ctf(b,0)  
                           , vec_ctf(c,0)  
                           ),0);
```

# The pack abstraction

---

## `simd::pack<T>`

`pack<T, N>`    SIMD register that packs N elements of type T  
`pack<T>`        automatically finds best N available

Behaves just like T except operations yield a pack of T and not a T.

## Constraints

- T must be a fundamental arithmetic type, i.e. `(un)signed char`, `(unsigned) short`, `(unsigned) int`, `(unsigned) long`, `(unsigned) long long`, `float` or `double`.
- `bool` support is done with `logical<T>`.
- N must be a power of 2.

# pack API

---

## Operators

- All overloadable operators are available :

`pack<T> ⊕ pack<T> , pack<T> ⊕ T , T ⊕ pack<T>`

- Type coercion and promotion disabled

`uint8_t(255) + uint8_t(1) yields uint8_t(0), not int(256)`

## Comparisons

- `==`, `!=`, `<`, `<=`, `>` and `>=` perform comparisons and return SIMD vectors.
- `compare_eq`, `compare_neq`, ... as functions return the result of the lexical comparison between the inputs as a `bool`.

## Other properties

- Models both a `RandomAccessFusionSequence` and `RandomAccessRange`
- `at_c<i>(p)` or `p[i]` can be used to access the *i*-th element

# pack API

---

## Memory access

- Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from/to a `T*`.
- Errors asserts in debug mode.

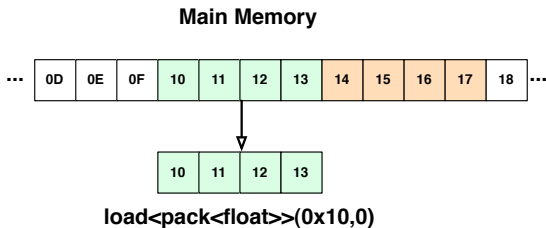
# pack API

## Memory access

- Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from/to a `T*`.
- Errors asserts in debug mode.

## Examples

`load< pack<T, N> >(p, i)` loads pack at aligned address `p + i`



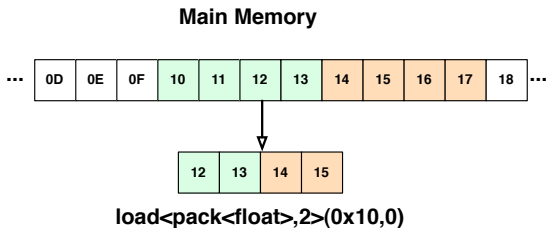
# pack API

## Memory access

- Memory must be aligned on `sizeof(T)*N` to load/store a `pack<T, N>` from/to a `T*`.
- Errors asserts in debug mode.

## Examples

`load< pack<T, N>, Offset>(p, i)` loads pack at address `p + i + Offset`, `p + i` must be aligned.



# Expression Template to the rescue !

---

## Rationale

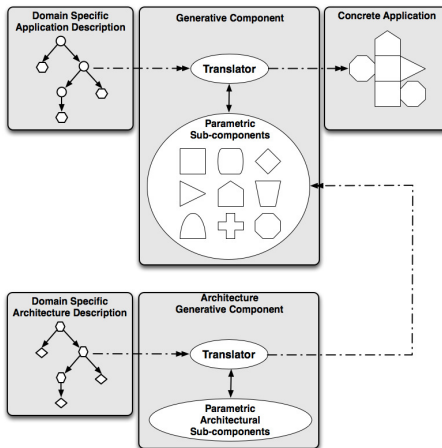
- Most SIMD ISA have fused operations (FMA, etc...)
- We want to write simple code but yet get best performances out of these
- We need lazy evaluation : Boost.Proto to the rescue !

## Advantage

- All expressions, even those involving functions, generate template expressions that are evaluated on assignment or in the conversion operator
- $a * b + c$  is mapped to `fma(a, b, c)`  
 $a + b * c$  is mapped to `fma(b, c, a)`  
 $!(a < b)$  is mapped to `is_nle(a, b)`
- The optimization system is open for extensions

# Expression Template to the rescue !

## Generative programming idioms used





# Extra arithmetic, bitwise and ieee operations, predicates

---

## Arithmetic

- saturated arithmetic
- float/int conversion
- round, floor, ceil, trunc
- sqrt, hypot
- average
- random
- min/max
- rounded division and remainder

## Bitwise

- select
- andnot, ornot
- popcnt
- ffs
- ror, rol
- rshr, rshl
- twopower

## IEEE

- ilogb, frexp

- ldexp
- next/prev
- ulpdist

## Predicates

- comparison with zero
- negation of comparison
- is\_unord, is\_nan, is\_invalid
- is\_odd, is\_even
- majority

# Reduction and SWAR operations

---

## Reduction

- any, all
- nbtrue
- minimum/maximum, posmin/posmax
- sum
- product, dot product

## SWAR

- group/split
- splatted reduction
- cumsum
- sort

# Benchmarks for floats

---

Timing performance for `float`(in cycles/value)

Architecture : Core I7 SandyBridge, AVX

| Function | Range             | STD or Other | Scalar | SIMD |
|----------|-------------------|--------------|--------|------|
| exp      | $[-10, 10]$       | 46           | 38     | 7    |
| log      | $[-10, -10]$      | 42           | 37     | 5    |
| asin     | $[-1, 1]$         | 40           | 35     | 13   |
| cos      | $[-20\pi, 20\pi]$ | 66           | 47     | 6    |
| fast_cos | $[-\pi/4, \pi/4]$ | 32           | 9      | 1.3  |

# RGB to grayscale

---

## Scalar version

```
float const *red, *green, *blue;
float* result;

for(std::size_t i = 0; i != height*width; ++i)
    result[i] = 0.3f * red[i] + 0.59f * green[i] + 0.11f * blue[i];
```

## SIMD version

```
auto rgb = fusion::make_vector(red, green, blue);
for(std::size_t i = 0; i != height*width; i+=pack<float>::static_size)
{
    auto p = load<decltype(rgb)>(rgb, i);
    auto res = 0.3f*fusion::at_c<0>(p)+0.59f*fusion::at_c<1>(p)
              +0.11f*fusion::at_c<2>(p);
    store(res, result, i);
}
```

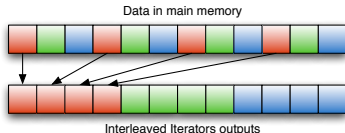
## Easy enough, but what if...

---

- ... I've got interleaved RGB or RGBA?

# Interleaved RGB

## What do we want to do?



## Problem

- On current SIMD extensions, this required a combination of unaligned load operations and vector shuffling.
- **Yes but gather is coming!** In AVX2, we will be able to gather nonadjacent data elements.

## Proposal

- We need to have a proper abstraction for deinterleaving data.

# Interleaved RGB

---

## Code example

```
float *data, *result; // Interleaved data start here.
simd::interleaved_iterator<float,3> b(data);
simd::interleaved_iterator<float,3> e(data+size);
std::transform(b, e, simd::begin(result)
               ,[](decltype(*b) v)
               {
                   return    0.3f *fusion::at_c<0>(v)
                          + 0.59f*fusion::at_c<1>(v)
                          + 0.11f*fusion::at_c<2>(v);
               }
               );
```

# Talk Layout

---

Introduction

Interface

boost.simd and the Standard Library

Results

Conclusion



# Operations vs Data

---

## Where/How to store our data ?

- SIMD operations require data to operate onto
- Usual approaches force a specific container type onto users
- Not generic enough

# Operations vs Data

---

## Where/How to store our data ?

- SIMD operations require data to operate onto
- Usual approaches force a specific container type onto users
- Not generic enough

## A better approach

- SIMD compliant allocators
- SIMD Range and Iterators over ContiguousRange
- Adapt our SIMD classes to work with a subset of STD algorithms

# SIMD allocators

---

## Rationale

- Allow containers to handle memory in a SIMD compliant way
- Handles alignment of memory
- Handles padding of memory

## Example

```
std::vector<float, simd::allocator<float> > v(173);  
  
assert( simd::is_aligned(&v[0]) );
```

# From Range to SIMDRange

---

## Iterator interface

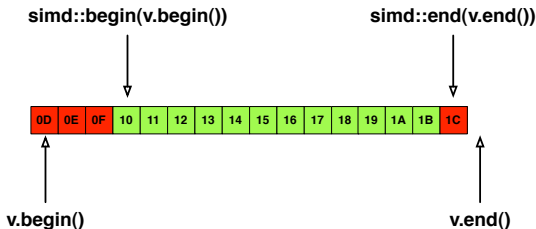
- Boost.SIMD provides `simd::begin()/simd::end()`
- Turn iterators into SIMD iterators returning pack
- Take a regular range, iterate over it in SIMD

# From Range to SIMDRange

## Iterator interface

- Boost.SIMD provides `simd::begin()/simd::end()`
- Turn iterators into SIMD iterators returning pack
- Take a regular range, iterate over it in SIMD

## Example



# From Range to SIMDRange

---

## Iterator interface

- Boost.SIMD provides `simd::begin()/simd::end()`
- Turn iterators into SIMD iterators returning pack
- Take a regular range, iterate over it in SIMD

## Example

```
std::vector<float, simd::allocator<float> > v(1024);  
pack<float> x,z;  
  
x = boost::accumulate(simd::range(v), z);
```

# From Range to SIMDRange

---

## Iterator interface

- pack provides `begin()/end()`
- Directly usable in STD algorithms
- Directly usable in Boost.Range algorithms

## Example

```
pack<float> x(1,2,3,4);
```

```
float k = std::accumulate(x.begin(), x.end(), 0.f);
```

## SIMD values as Range

---

### Putting everything together

```
std::vector<float, simd::allocator<float> > v(1024);  
pack<float> x,z;  
float r;
```

```
x = boost::accumulate(simd::range(v), z);  
r = std::accumulate(x.begin(), x.end(), 0.f);
```

### The proper way

```
std::vector<float, simd::allocator<float> > v(1024);  
pack<float> x,z;  
float r;
```

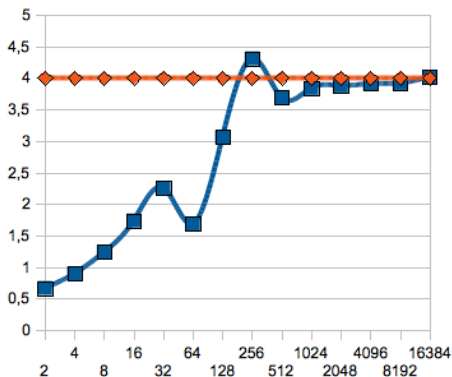
```
r = simd::accumulate( simd::range(v), pack<float>() );
```



# SIMD values as Range

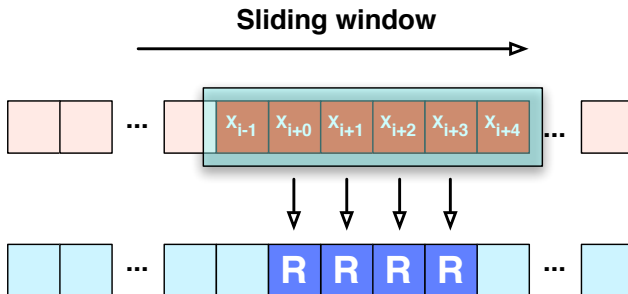
`std::accumulate` speed-up for float

Architecture: Core2Duo Arrandale, SSE2.



# An other example : SIMD Filtering

## Unidimensional filtering



$$VR = 1/3 * ( \text{load}<-1>(vx) + \text{load}<0>(vx) + \text{load}<1>(vx) )$$

## Another example : SIMD Filtering

---

### Generic SIMD/scalar code

```
template<class RangeIn, class RangeOut>
inline void average( RangeOut result, RangeIn input )
{
    typedef typename RangeIn::iterator in_iterator;
    typedef typename RangeOut::iterator iterator;
    typedef typename iterator_value<iterator>::type type;

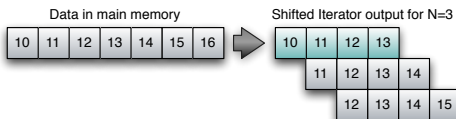
    iterator br = result.begin(), er = result.end();
    in_iterator data = input.begin();

    br++; er--;
    while( br != er )
    {
        type xm1 = load<type, -1>(data,i);
        type x   = load< type >(data,i);
        type xp1 = load<type, +1>(data,i);
        res = = 1.f/3 * (xm1 + x + xp1);
        store(res,i,0);
    }
}
```

# An other example : SIMD Filtering

## Unidimensional filtering

- Hum... seems like something recurrent in the image/signal processing world.



## Proposal

- Abstract the sliding window computation pattern with a `shifted_iterator`.

## Another example : SIMD Filtering

---

Looks like this !

```
float *data, *result;
simd::shifted_iterator<float,3> b(data);
simd::shifted_iterator<float,3> e(data+size);
std::transform(b, e, simd::begin(result)
               ,[](decltype(*b) v)
               {
                   return 1.f/3*(v[0]+v[1]+v[2]);
               }
               );
```

# Talk Layout

---

Introduction

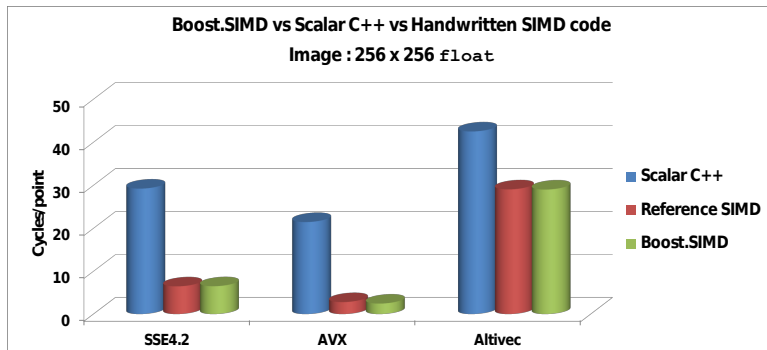
Interface

boost.simd and the Standard Library

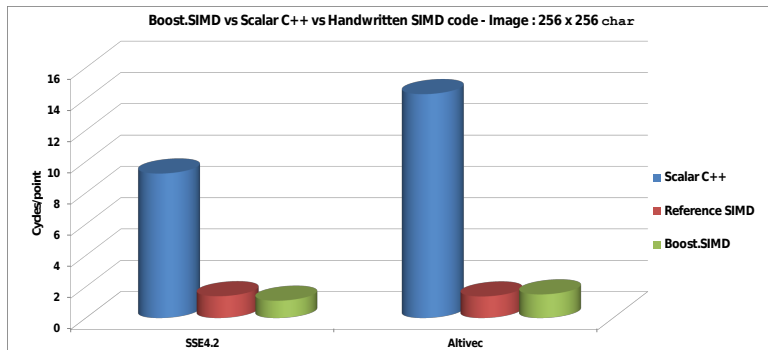
Results

Conclusion

# RGB2YUV



# Sigma Delta Motion detection Algorithm





# Thomas Tri-diagonal Solver

---

Algorithm 1: Thomas algorithm: solve  $Ax=s$ .

---

input : A tridiagonal system.

- $a[n]$  : sub-diagonal
- $b[n]$  : main diagonal
- $c[n]$  : sup-diagonal
- $s[n]$  : right hand side

begin

Forward elimination:

for  $i=2,...,n$  do

$\delta = a[i]c[i-1]/b[i-1]$

$b[i] = b[i] - \delta$

$\delta_s = a[i]s[i-1]/b[i-1]$

$s[i] = s[i] - \delta_s$

end

Backward substitution:  $x[n] = s[n]/b[n]$

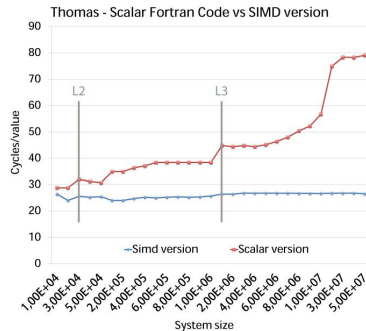
for  $i=n-1,...,1$  do

$x[i] = s[i] - c[i]x[i+1]/b[i]$

end

end

---



# Talk Layout

---

Introduction

Interface

boost.simd and the Standard Library

Results

Conclusion

# Current Work

---

## Integrated SIMD support

- Most STD algorithms should be specialized to be run in one scope
- Can we have a `Boost.Range` adaptor like `simd(r)` ?
- Support for shifted Range using `load<T,N>` and unaligned Range

## What's on Fire ?

- ISA improvements for ARM, MIC
- Submission to Boost C++ Libraries... soon...

# Overview of Boost.SIMD

---

## Our goals

- Bring SIMD programming to a usable state
- If we have `Boost.Atomic`, why not `Boost.SIMD` ?
- Be attractive by being nice with the rest of C++

## What we achieved

- Demonstrated some impacts in term of performance
- Made using SIMD almost as simple than scalar

# Thanks for your attention